Metro Rail Management System

Table of Contents

1. **Requirement Analysis**

1.1 Functional Requirements

This project can perform the following:

1. **Input/Output**: To calculate the fare of a trip, the source and destination and provided as input and the system provides the evaluated fare for the trip based on a multiplicative factor.
2. **Processing**: The system implements a custom foreign key which checks for the parent key in two tables, namely Lines and Crossing.

   The system also has the capability to maintain integrity and will automatically delete every stations from the database if the line having those database is removed. When stations are removed all outdated routes are removed alongside so that an invalid route does not get assigned to a train. Upon removal of routes all train which currently follow the removed route are also removed such that a train does not run on a non-existing route.

   On new insertion of stations the system asks the DB_ADMIN to specify the line on which the station is present and on which end of the line is it going to be inserted. The system then accordingly creates a new station wherever required such as the Line_XY table and updates the sequence of stations accordingly.

   The system also create a new table for a new line specifying the stations which will be present on that line. **** The system also has the ability to find crossing stations from the tables and give them a new line_id from the crossings table. Whenever an existing station is inserted into a line_XY table a procedure is called which check for the already existing station in all other line_xy tables and if it is found it is branded as a crossing station and its line_id is updated in the stations table from the crossings table.

3. **Error Handling**: System reports any errors on duplicate primary keys and also report any ‘Out of Range’ values on numeric fields. The system provides system application error for custom foreign key and for out of scope price calculation. If methods other than the ones recommended are used to interact with the database, the database will exhibit undefined behaviour which may be fatal.

1.2 Feasibility Analysis

1. **Technical Feasibility**: The proposed system is technically feasible because it can be easily implemented using SQL. We are using the resources which are already available.
2. **Operational Feasibility**: The proposed system is also operationally feasible because it would be easy to use if instructions are followed and the user is skilled enough to use the proposed system.
3. **Legal Feasibility**: The proposed system is totally legally feasible. No infringement is included in the system. Legal aspects and government regulations have been taken in notice.
4. **Schedule Feasibility**: The Metrorail management system could be developed in a reasonable amount of time up to which the client is satisfied.
5. **Cultural Feasibility**: The proposed system is culturally feasible. The system will be easily accepted by the organization. It does not affect any scientific as well as ethical, behavioural, and social issue
6. **Economic Feasibility**: The proposed system is economically feasible because it reduces the operational cost which would occur if the system is not present. The system initially requires high speed hardware to ensure fast processing.
7. **Behavioural feasibility**: This technique is user friendly.

1.3 Other Non-Functional Requirements

1. Performance Requirements: The system may perform poorly if the system is not run on a high speed hardware. This a real-time system, so performance is of utmost importance. The consistency and integrity of data is taken into consideration for better performance.

2. Safety Requirements: Another thing to be taken care of is the back-up of the database. If in any case the database becomes corrupt or data gets lost, we must have a proper back-up of our data.

3. Security Requirements: Currently the system is based around a central database, thus it is secure from online attacks as its is offline.

2. Entity Relationship Diagrams

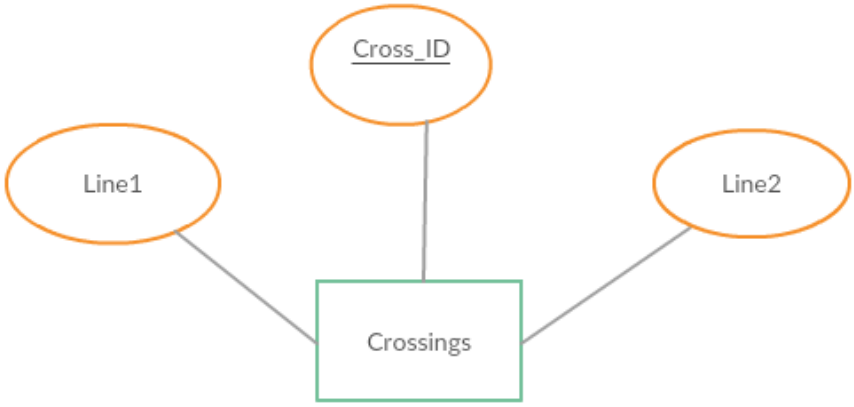


Figure 1 Crossing E-R Diagram

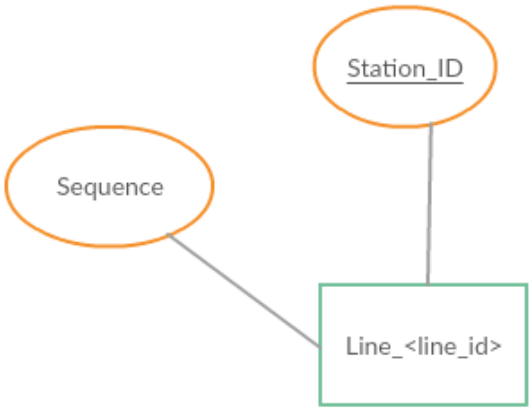


Figure 2 Line_XY E-R Diagram

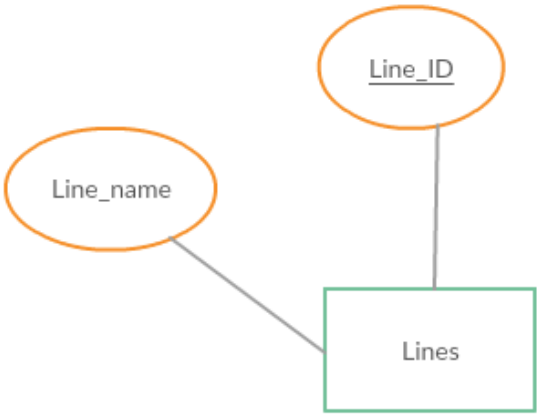


Figure 3 Lines E-R Diagram

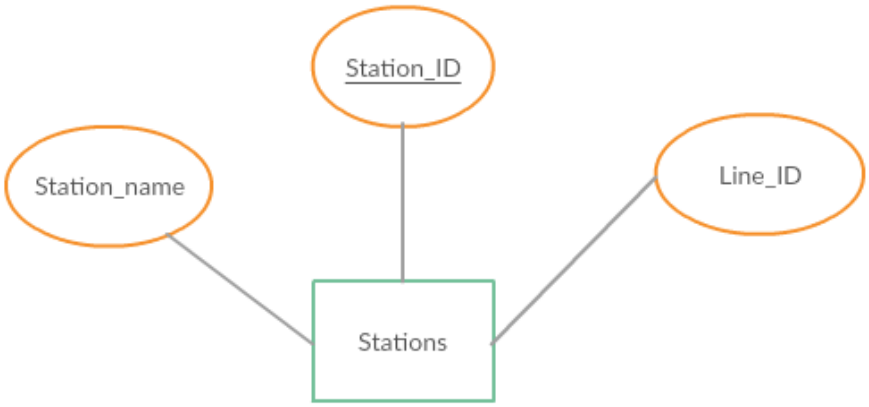


Figure 4 Stations E-R Diagram

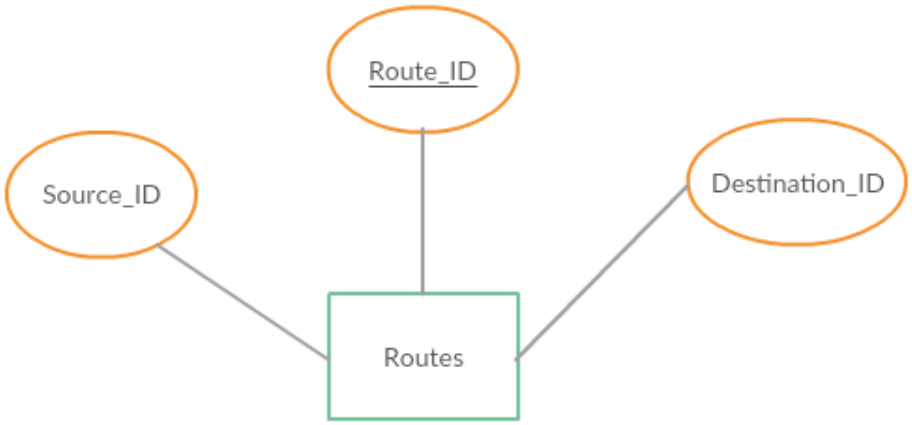


Figure 5 Routes E-R Diagram

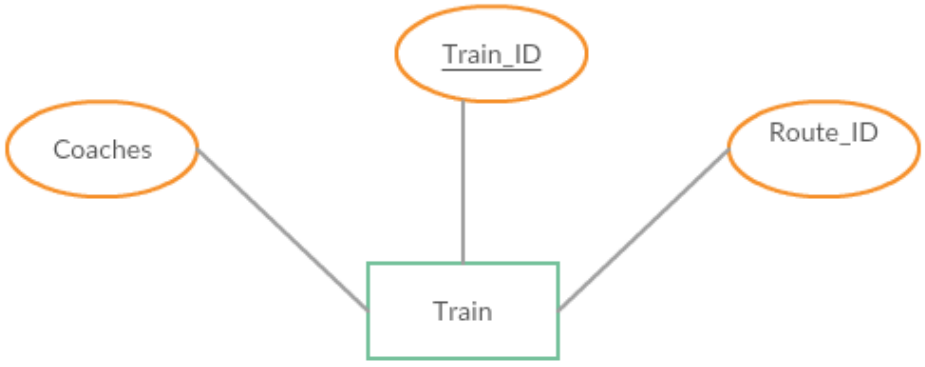


Figure 6 Train E-R Diagram

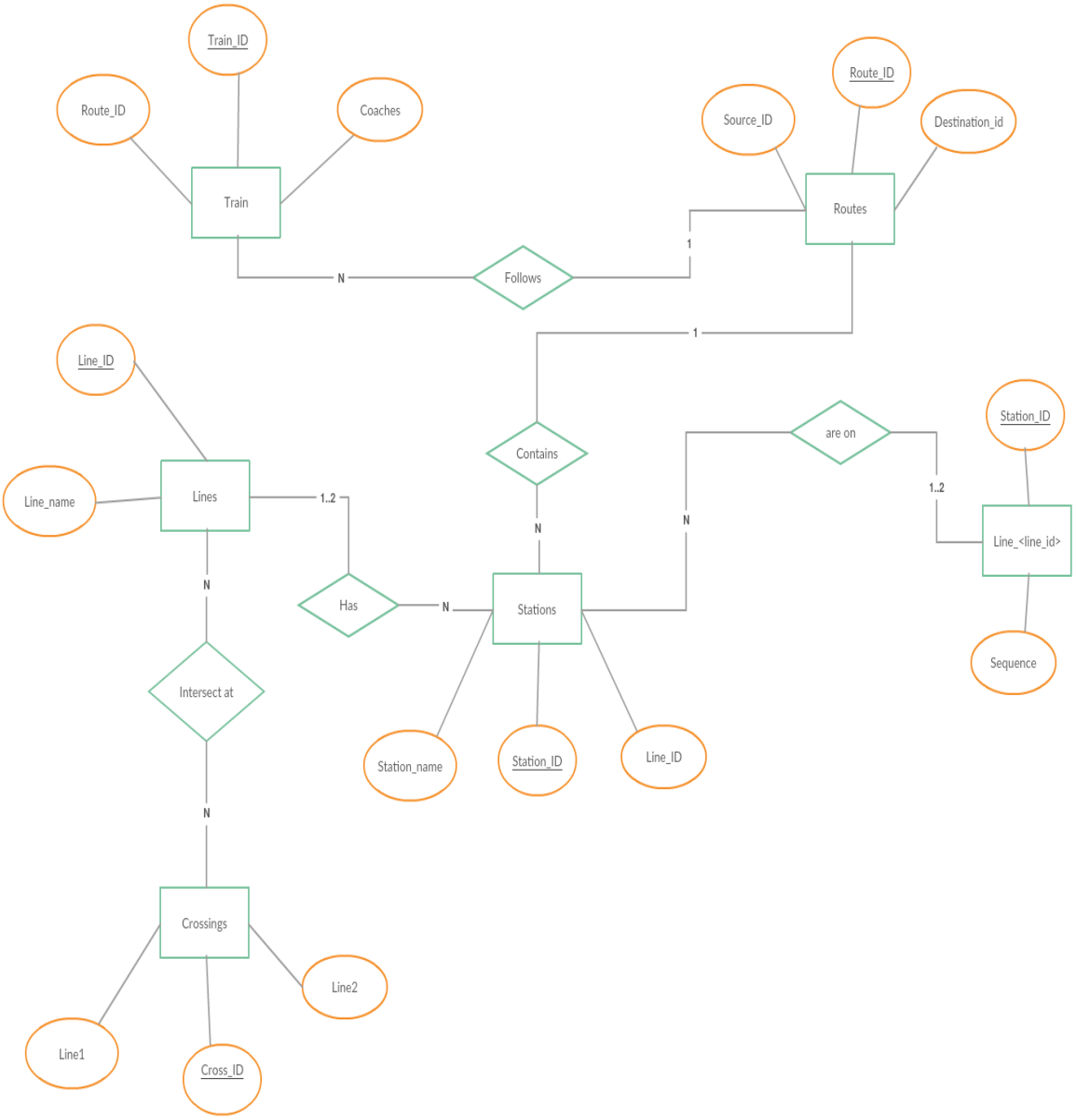


Figure 7 Database E-R Diagram

3. Database Tables:

Table 1 Crossing Table

Name	Null?	Type
CROSS_ID	NOT NULL	NUMBER
LINE1	NOT NULL	NUMBER
LINE2	NOT NULL	NUMBER

Table 2 Line_XY

Name	Null?	Type
STATION_ID	NOT NULL	NUMBER
SEQ	NOT NULL	NUMBER

Table 3 Lines Table

Name	Null?	Type
LINE_ID	NOT NULL	NUMBER
LINE_NAME	NOT NULL	VARCHAR2(45)

Table 4 Routes Table

Name	Null?	Type
ROUTE_ID	NOT NULL	NUMBER
SOURCE_ID	NOT NULL	NUMBER
DEST_ID	NOT NULL	NUMBER

Table 5 Stations Table

Name	Null?	Type
STATION_ID	NOT NULL	NUMBER
STATION_NAME	NOT NULL	VARCHAR2(45)
LINE_ID	NOT NULL	NUMBER

Table 6 Train Table

Name	Null?	Type
TRAIN_ID	NOT NULL	NUMBER
ROUTE_ID	NOT NULL	NUMBER
COACHES	NOT NULL	NUMBER

4. Validation of Database with Normalization

The database is in 1NF, because it does not have any attribute which can be decomposed to create multiple attributes.

The database is in 2NF, because all non-key attributes of any given table are fully functional dependent on the primary key.

Eg. In the Train table, the non-key attributes –coaches and route_id are fully functional dependent on the primary key-train_id.

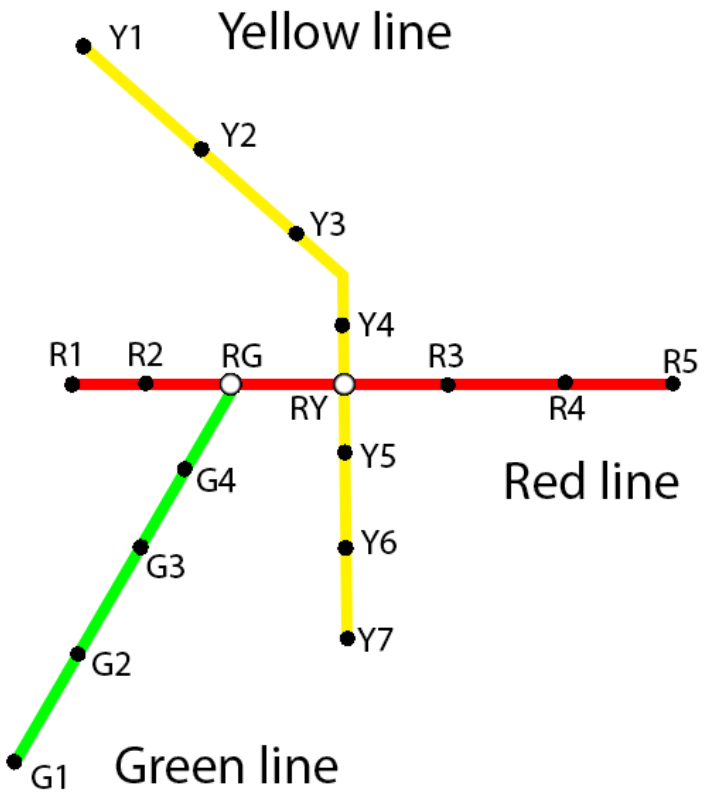
The database is 3NF, because no nonprime attribute of a table is transitively dependent on another the key.

The database is also 4NF, as it contains no multi-valued dependencies i.e. no attribute is a ‘multi-valued fact’ about another.

Normalization assists in removing unnecessary redundancies from the database and helps improve performance and response time of the system.

5. INPUT/ OUTPUT

The following diagram has been taken as a sample input for the database:



The expected table output is:

Table 7: Lines

Line_ID	Line_Name
1	Yellow
2	Green
4	Red

Table 8: Crossings

Cross_ID	Line1	Line2
3	2	1
5	4	1

6	4	2

Table 9: Stations

Station_ID	Station_Name	Line_ID
1	Y1	1
2	Y2	1
3	Y3	1
4	Y4	1
5	RY	5
6	Y5	1
7	Y6	1
8	Y7	1
9	G1	2
10	G2	2
11	G3	2
12	G4	2
13	RG	6
14	R1	4
15	R2	4
16	R3	4
17	R4	4
18	R5	4

Table 10: Line_1

Station_ID	Seq
1	1
2	2
3	3
4	4
5	5
6	6
7	7

8	8

Table 11: Line_2

Station_ID	Seq
9	1
10	2
11	3
12	4
13	5

Table 12 Line_4

Station_ID	Seq
14	1
15	2
13	3
5	4
16	5
17	6
18	7

Table 13 Routes

Route_ID	Source_ID	Destination_ID
1	14	18
2	18	14
3	1	8
4	8	1
5	9	13
6	13	9
7	14	5
8	8	5

Table 14 Train

Train_ID	Route_ID	Coaches
1	7	8
2	5	4
3	1	6
4	8	8
5	3	8

6. PL/SQL Code:

```
-- ------------------------------------------------------
-- Table Stations
-- ------------------------------------------------------
CREATE TABLE Stations (
    Station_ID NUMBER NOT NULL,
    Station_Name VARCHAR2(45) UNIQUE NOT NULL,
    Line_ID NUMBER NOT NULL,
    PRIMARY KEY (Station_ID));

-- ------------------------------------------------------
-- Table Routes
-- ------------------------------------------------------
CREATE TABLE Routes (
    Route_ID NUMBER NOT NULL,
    Source_ID NUMBER NOT NULL,
    Dest_ID NUMBER NOT NULL,
    PRIMARY KEY (Route_ID),
    CONSTRAINT FK_SOURCE_ID FOREIGN KEY(Source_ID) REFERENCES
Stations(Station_ID),
    CONSTRAINT FK_DEST_ID FOREIGN KEY(Dest_ID) REFERENCES
Stations(Station_ID));

-- ------------------------------------------------------
-- Table Train
-- ------------------------------------------------------
CREATE TABLE Train (
    Train_ID NUMBER NOT NULL,
    Route_ID NUMBER NOT NULL,
    Coaches NUMBER NOT NULL,
    PRIMARY KEY (Train_ID),
    CONSTRAINT CHECK_COACHES CHECK (Coaches IN (4,6,8)),
    CONSTRAINT FK_ROUTE_ID FOREIGN KEY(Route_ID) REFERENCES
Routes(Route_ID));

-- ------------------------------------------------------
-- Table Lines
-- ------------------------------------------------------
CREATE TABLE Lines (
    Line_ID NUMBER NOT NULL,
    Line_Name VARCHAR2(45) UNIQUE NOT NULL,
    PRIMARY KEY (Line_ID));

-- ------------------------------------------------------
-- Table Crossings
-- ------------------------------------------------------
```

```
CREATE TABLE Crossings (
    Cross_ID NUMBER NOT NULL,
    Line1 NUMBER NOT NULL,
    Line2 NUMBER NOT NULL,
    PRIMARY KEY (Cross_ID),
    CONSTRAINT FK_CROSSINGS_LINE1 FOREIGN KEY(Line1) REFERENCES
Lines(Line_ID),
    CONSTRAINT FK_CROSSINGS_LINE2 FOREIGN KEY(Line2) REFERENCES
Lines(Line_ID));

-- ------------------------------------------------------
-- Table LineXY
-- ------------------------------------------------------
CREATE TABLE LINE_<LINE_ID> (
    Station_ID NUMBER NOT NULL,
    Seq NUMBER NOT NULL,
    PRIMARY KEY (Station_ID),
    CONSTRAINT FK_LINEN_Station_ID FOREIGN KEY(Station_ID) REFERENCES
Stations(Station_ID));

-- ------------------------------------------------------
-- Trigger for FOREIGN KEY on Line_ID on table Stations
-- ------------------------------------------------------
CREATE OR REPLACE TRIGGER Line_ID_FK
BEFORE INSERT OR UPDATE OF Line_ID ON Stations
FOR EACH ROW
DECLARE
    CURSOR LINES IS SELECT LINE_ID FROM Lines;
    CURSOR CROSSINGS IS SELECT Cross_ID FROM Crossings;
    FOUND EXCEPTION;
    A LINES%ROWTYPE;
BEGIN
    FOR A IN LINES LOOP
        IF :NEW.LINE_ID = A.Line_ID THEN
            RAISE FOUND;
        END IF;
    END LOOP;
    FOR A IN CROSSINGS LOOP
        IF :NEW.LINE_ID = A.Cross_ID THEN
            RAISE FOUND;
        END IF;
    END LOOP;
    RAISE_APPLICATION_ERROR(-20045,'FOREIGN KEY VIOLATED BECAUSE Station_ID
NOT FOUND IN PARENT TABLES');
EXCEPTION
    WHEN FOUND THEN
        NULL;
```

```
END;

-- ------------------------------------------------------------
-- Trigger for CASCADE DELETE on Line_ID removal on table Lines
-- ------------------------------------------------------------
CREATE OR REPLACE TRIGGER Line_ID_cascade_delete
AFTER DELETE ON Lines
FOR EACH ROW
DECLARE
    CURSOR CROSS1 IS SELECT * FROM Crossings WHERE Line1 = :OLD.Line_ID OR
Line2 = :OLD.Line_ID;
    ROW1 Crossings%ROWTYPE;
    VAL Lines.Line_ID%TYPE;
BEGIN
    -- DELETE FROM TABLE Stations, WITH LINE_ID
    DELETE FROM Stations WHERE Line_ID = :OLD.Line_ID;
    -- UPDATE TABLE Stations, WHERE CROSSINGS WITH LINE_ID, AND DELETE
CROSSINGS ENTRY
    FOR ROW1 IN CROSS1 LOOP
        IF ROW1.Line1 = :OLD.Line_ID THEN
            VAL := ROW1.Line2;
        ELSE
            VAL := ROW1.Line1;
        END IF;
        UPDATE Stations SET Line_ID = VAL WHERE Line_ID = ROW1.Cross_ID;
        DELETE FROM Crossings WHERE Cross_ID = ROW1.Cross_ID;
    END LOOP;
    -- DROP TABLE LINE_LINE_ID
        EXECUTE IMMEDIATE 'DROP TABLE LINE_'||:OLD.Line_ID;
END;

-- ------------------------------------------------------------
-- Trigger for CASCADE DELETE OF routes on delete of Station
-- ------------------------------------------------------------
CREATE OR REPLACE TRIGGER Station_cascade_delete
AFTER DELETE ON Stations
FOR EACH ROW
BEGIN
    DELETE FROM Routes WHERE Source_ID = :OLD.Station_ID OR Dest_ID =
:OLD.Station_ID;
    -- UPDATE VALUES OF OTHER STATIONS
    UPDATE Stations SET Station_ID = Station_ID-1 WHERE Station_ID >
:OLD.Station_ID;
END;

-- ------------------------------------------------------------
-- Trigger for CASCADE DELETE OF Trains ON DELETE OF Routes
```

```
-- ------------------------------------------------------
CREATE OR REPLACE TRIGGER Route_cascade_delete
AFTER DELETE ON Routes
FOR EACH ROW
BEGIN
    DELETE FROM Train WHERE Route_ID = :OLD.Route_ID;
END;

-- ------------------------------------------------------
-- TRIGGER ON UPDATE OF STATION_ID ON STATIONS
-- ------------------------------------------------------
CREATE OR REPLACE TRIGGER UPDATE_STATION_ID
AFTER UPDATE OF Station_ID ON Stations
FOR EACH ROW
DECLARE
    VAL Stations.Line_ID%TYPE;
    VAL1 Stations.Line_ID%TYPE;
BEGIN
    UPDATE Routes SET Source_ID = :NEW.Station_ID WHERE Source_ID =
:OLD.Station_ID;
    UPDATE Routes SET Dest_ID = :NEW.Station_ID WHERE Dest_ID =
:OLD.Station_ID;
    SELECT COUNT(*) INTO VAL FROM Lines WHERE Line_ID = :NEW.Line_ID;
    IF VAL > 0 THEN
        -- UPDATE IN LINEXY
        EXECUTE IMMEDIATE 'UPDATE LINE_'||:NEW.Line_ID||' SET Station_ID =
'||:NEW.Station_ID||' WHERE Station_ID = '||:OLD.Station_ID;
    ELSE
        -- RETRIEVE BOTH LINES AND UPDATE IN BOTH
        SELECT Line1, Line2 INTO VAL, VAL1 FROM Crossings WHERE Cross_ID =
:OLD.Line_ID;
        EXECUTE IMMEDIATE 'UPDATE LINE_'||VAL||' SET Station_ID =
'||:NEW.Station_ID||' WHERE Station_ID = '||:OLD.Station_ID;
        EXECUTE IMMEDIATE 'UPDATE LINE_'||VAL1||' SET Station_ID =
'||:NEW.Station_ID||' WHERE Station_ID = '||:OLD.Station_ID;
    END IF;
END;

-- ------------------------------------------------------
-- PROCEDURE TO CREATE LINEXY ON INSERT ON Lines
-- ------------------------------------------------------
CREATE OR REPLACE PROCEDURE TRIG_INSERT_Lines (LINEID Lines.Line_ID%TYPE)
AS
    CURSOR LINES IS SELECT Line_ID FROM Lines WHERE Line_ID <> LINEID;
    A LINES%ROWTYPE;
    VAL NUMBER;
    VAL1 NUMBER;
```

```
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE LINE_'||LINEID||'(Station_ID NUMBER NOT
NULL,Seq NUMBER NOT NULL, PRIMARY KEY (Station_ID),CONSTRAINT
FK_LINE'||LINEID||'_Station_ID FOREIGN KEY(Station_ID) REFERENCES
Stations(Station_ID))';
    -- CREATE TRIGGER FOR LINE_LINE_ID
    EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER INSERT_LINE'||LINEID||'
AFTER INSERT ON LINE_'||LINEID||' FOR EACH ROW BEGIN
CHECK_FOR_CROSSINGS('||LINEID||', :NEW.Station_ID);END;';
    -- CREATE ENTRIES IN TABLE CROSSINGS
    SELECT COUNT(*) INTO VAL FROM Lines;
    IF VAL > 0 THEN
        --CREATE COMBINATIONS
        VAL1 := LINEID;
        FOR A IN LINES LOOP
            VAL1 := VAL1+1;
            INSERT INTO Crossings VALUES (VAL1,A.Line_ID, LINEID);
        END LOOP;
    END IF;
END;

-- ----------------------------------------------------------------------
------------
-- PROCEDURE TO CHECK IF STATION IS A CROSSING, AND ACCORDINGLY MAKE AMENDS
IN Stations
-- ----------------------------------------------------------------------
------------
CREATE OR REPLACE PROCEDURE CHECK_FOR_CROSSINGS (LINE Lines.Line_ID%TYPE,
STATION Stations.Station_ID%TYPE)
AS
    CURSOR LINES IS SELECT Line_ID FROM Lines WHERE Line_ID <> LINE;
    A LINES%ROWTYPE;
    VAL NUMBER;
BEGIN
    FOR A IN LINES LOOP
        -- SEARCH IN LINE_XY FOR STATION
        EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM LINE_'||A.Line_ID||' WHERE
Station_ID = '||STATION INTO VAL;
        IF VAL > 0 THEN
            SELECT Cross_ID INTO VAL FROM Crossings WHERE (Line1 = LINE AND
Line2 = A.Line_ID) OR (Line2 = LINE AND Line1 = A.Line_ID);
            UPDATE Stations SET Line_ID = VAL WHERE Station_ID = STATION;
        END IF;
    END LOOP;
END;

-- ----------------------------------------------------
```

```
-- FUNCTION TO CALCULATE PRICE BETWEEN TWO STATIONS
-- ------------------------------------------------
CREATE OR REPLACE FUNCTION CALC_PRICE (STAT1 Stations.Station_ID%TYPE,
STAT2 Stations.Station_ID%TYPE) RETURN NUMBER AS

    LINE1 Lines.Line_ID%TYPE;
    LINE2 Lines.Line_ID%TYPE;
    TOO_FAR EXCEPTION;
    MULTI CONSTANT NUMBER := 2;
    MINVAL CONSTANT NUMBER := 8;
    MAXVAL CONSTANT NUMBER := 60;
    VAL1 NUMBER;
    VAL2 NUMBER;
    V1 NUMBER;
    V2 NUMBER;
    V3 NUMBER;
    V4 NUMBER;
    CALC NUMBER;

    FUNCTION SIMPLE_CALC (STAT1 Stations.Station_ID%TYPE, STAT2
Stations.Station_ID%TYPE, LINE NUMBER) RETURN NUMBER AS
    RES NUMBER;
    V1 NUMBER;
    V2 NUMBER;
    BEGIN
        EXECUTE IMMEDIATE 'SELECT Seq FROM LINE_'||LINE||' WHERE Station_ID
= '||STAT1 INTO V1;
        EXECUTE IMMEDIATE 'SELECT Seq FROM LINE_'||LINE||' WHERE Station_ID
= '||STAT2 INTO V2;
        IF V1 > V2 THEN
            RES := V1 - V2;
        ELSE
            RES := V2 - V1;
        END IF;
        RETURN (RES);
    END;

    FUNCTION FIND_COMMON_LINE (V1 NUMBER, V2 NUMBER, V3 NUMBER, V4 NUMBER)
RETURN NUMBER AS
    BEGIN
        IF V1 <> V3 AND V1 <> V4 AND V2 <>V3 AND V2 <>V4 THEN
            RETURN(0);
        ELSIF V1 = V3 THEN
            RETURN(V1);
        ELSIF V1 = V4 THEN
            RETURN(V1);
        ELSIF V2 = V3 THEN
```

```
            RETURN(V2);
        ELSIF V2 = V4 THEN
            RETURN(V2);
        END IF;
    END;

    FUNCTION FIND_CROSSING (LINE1 NUMBER, LINE2 NUMBER) RETURN NUMBER AS
    STATI Stations.Station_ID%TYPE;
    LINE NUMBER;
    BEGIN
        SELECT Cross_ID INTO LINE FROM Crossings WHERE (Line1 = LINE1 AND
Line2 = LINE2) OR (Line1 = LINE2 AND Line2 = LINE1);
        SELECT Station_ID INTO STATI FROM Stations WHERE Line_ID = LINE;
        RETURN(STATI);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN(0);
        WHEN TOO_MANY_ROWS THEN
            RETURN(0);
    END;
BEGIN
    CALC := 0;
    SELECT Line_ID INTO LINE1 FROM Stations WHERE Station_ID = STAT1;
    SELECT Line_ID INTO LINE2 FROM Stations WHERE Station_ID = STAT2;
    SELECT COUNT(*) INTO VAL1 FROM Lines WHERE Line_ID = LINE1;
    SELECT COUNT(*) INTO VAL2 FROM Lines WHERE Line_ID = LINE2;

    IF LINE1 = LINE2 THEN
        -- SIMPLY CALCULATE DIFFERENCE IN STATIONS ON THE LINE
            CALC := SIMPLE_CALC(STAT1, STAT2, LINE1);
    ELSE
        -- CHECK IF BOTH ON DIFFERENT LINE OR SAME
        IF VAL1 = 0 AND VAL2 = 0 THEN
            -- BOTH ARE CROSSINGS, WITH ATLEAST ONE DIFFERENT LINE
            SELECT Line1, Line2 INTO V1,V2 FROM Crossings WHERE Cross_ID =
LINE1;
            SELECT Line1, Line2 INTO V3,V4 FROM Crossings WHERE Cross_ID =
LINE2;
            VAL1 := FIND_COMMON_LINE(V1,V2,V3,V4);
            IF VAL1 = 0 THEN
                -- NO COMMON LINE
                VAL1 := FIND_CROSSING(V1,V3);
                VAL2 := FIND_CROSSING(V1,V4);
                IF VAL1 <> 0 AND VAL2 <> 0 THEN
                    VAL1 := SIMPLE_CALC(STAT1, VAL1, V1)+SIMPLE_CALC(STAT2,
VAL1, V3);
```

```
                    VAL2 := SIMPLE_CALC(STAT1, VAL2, V1)+SIMPLE_CALC(STAT2,
VAL2, V4);
                    IF VAL1 <= VAL2 THEN
                        CALC := VAL1;
                    ELSE
                        CALC := VAL2;
                    END IF;
                ELSIF VAL1 <> 0 THEN
                    CALC := SIMPLE_CALC(STAT1, VAL1, V1)+SIMPLE_CALC(STAT2,
VAL1, V3);
                ELSIF VAL2 <> 0 THEN
                    CALC := SIMPLE_CALC(STAT1, VAL2, V1)+SIMPLE_CALC(STAT2,
VAL2, V4);
                END IF;
                VAL1 := FIND_CROSSING(V2,V3);
                VAL2 := FIND_CROSSING(V2,V4);
                IF VAL1 <> 0 THEN
                    IF CALC = 0 OR CALC > VAL1 THEN
                        CALC := VAL1;
                    END IF;
                END IF;
                IF VAL1 <> 0 THEN
                    IF CALC = 0 OR CALC > VAL2 THEN
                        CALC := VAL2;
                    END IF;
                END IF;
                IF CALC = 0 THEN
                    RAISE TOO_FAR;
                END IF;
            ELSE
                -- VAL1 IS COMMON LINE
                CALC := SIMPLE_CALC(STAT1, STAT2, VAL1);
            END IF;
        ELSIF VAL1 = 1 AND VAL2 = 1 THEN
            -- BOTH ARE ON DIFFERENT LINES
            V1 := FIND_CROSSING(LINE1,LINE2);
            IF V1 = 0 THEN
                RAISE TOO_FAR;
            END IF;
            CALC := SIMPLE_CALC(STAT1, V1,LINE1)+SIMPLE_CALC(STAT2,
V1,LINE2);
        ELSE
            -- ONE IS A CROSSING
            IF VAL1 = 0 THEN
                -- STAT1 IS A CROSSING
                SELECT Line1, Line2 INTO V1, V2 FROM Crossings WHERE
Cross_ID = LINE1;
```

```
                IF V1 = LINE2 THEN
                    CALC := SIMPLE_CALC(STAT1, STAT2, V1);
                ELSIF V2 = LINE2 THEN
                    CALC := SIMPLE_CALC(STAT1, STAT2, V2);
                ELSE
                    -- NO LINE COMMON
                    V3 := FIND_CROSSING(V1,LINE2);
                    V4 := FIND_CROSSING(V2,LINE2);
                    IF V3 = 0 AND V4 = 0 THEN
                        -- NO CROSSING FOUND
                        RAISE TOO_FAR;
                    ELSIF V3 = 0 THEN
                        -- V4 IS THE CROSSING
                        CALC := SIMPLE_CALC(STAT1,V4,V2) +
SIMPLE_CALC(STAT2,V4,LINE2);
                    ELSE
                        -- V3 IS THE CROSSING
                        CALC := SIMPLE_CALC(STAT1,V3,V1) +
SIMPLE_CALC(STAT2,V3,LINE2);
                    END IF;
                END IF;
            ELSIF VAL2 = 0 THEN
                -- STAT2 IS A CROSSING
                SELECT Line1, Line2 INTO V1, V2 FROM Crossings WHERE
Cross_ID = LINE2;
                IF V1 = LINE1 THEN
                    CALC := SIMPLE_CALC(STAT1, STAT2, V1);
                ELSIF V2 = LINE1 THEN
                    CALC := SIMPLE_CALC(STAT1, STAT2, V2);
                ELSE
                    -- NO LINE COMMON
                    V3 := FIND_CROSSING(V1,LINE1);
                    V4 := FIND_CROSSING(V2,LINE1);
                    IF V3 = 0 AND V4 = 0 THEN
                        -- NO CROSSING FOUND
                        RAISE TOO_FAR;
                    ELSIF V3 = 0 THEN
                        -- V4 IS THE CROSSING
                        CALC := SIMPLE_CALC(STAT1,V4,LINE1) +
SIMPLE_CALC(STAT2,V4,V2);
                    ELSE
                        -- V3 IS THE CROSSING
                        CALC := SIMPLE_CALC(STAT1,V3,LINE1) +
SIMPLE_CALC(STAT2,V3,V1);
                    END IF;
                END IF;
            END IF;
```

```
        END IF;
    END IF;
    CALC := MULTI*CALC;
    IF CALC < MINVAL THEN
        CALC := MINVAL;
    ELSIF CALC > MAXVAL THEN
        CALC := MAXVAL;
    END IF;
    RETURN(CALC);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-12004,'STATION ID NOT FOUND IN DATABASE');
    WHEN TOO_FAR THEN
        RAISE_APPLICATION_ERROR(-12005,'PRICE CALCULATION TOO COMPLEX,
BEYOND CURRENT SCOPE');
END;

-- ------------------------------------------------------
-- PROCEDURE TO INSERT ON LineXY
-- ------------------------------------------------------
CREATE OR REPLACE PROCEDURE INSERT_LINEXY(STATION
Stations.Station_Name%TYPE, LINE Lines.Line_Name%TYPE, AtEnd NUMBER)
AS
    LINEID Lines.Line_ID%TYPE;
    STATID Stations.Station_ID%TYPE;
    VAL NUMBER;
BEGIN
    SELECT Line_ID INTO LINEID FROM Lines WHERE LOWER(Line_Name) =
LOWER(Line);
    SELECT Station_ID INTO STATID FROM Stations WHERE LOWER(Station_Name) =
LOWER(STATION);
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM LINE_'||LINEID INTO VAL;
    IF AtEnd > 0 THEN
        IF VAL > 0 THEN
            EXECUTE IMMEDIATE 'SELECT MAX(Seq)+1 FROM LINE_'||LINEID INTO
VAL;
        ELSE
            VAL := 1;
        END IF;
    ELSE
        EXECUTE IMMEDIATE 'UPDATE LINE_'||LINEID||' SET Seq = Seq + 1';
        VAL := 1;
    END IF;
    EXECUTE IMMEDIATE 'INSERT INTO LINE_'||LINEID||' VALUES
('||STATID||','|| VAL||')';
END;
```

```
-- ------------------------------------------------------
-- PROCEDURE TO INSERT ON Lines
-- ------------------------------------------------------
CREATE OR REPLACE PROCEDURE INSERT_Lines (NAME Lines.Line_Name%TYPE)
AS
    V1 NUMBER;
BEGIN
    SELECT COUNT(*) INTO V1 FROM Lines;
    IF V1 = 0 THEN
        V1 := 1;
    ELSIF V1 = 1 THEN
        V1 := 2;
    ELSE
        SELECT MAX(Cross_ID) INTO V1 FROM Crossings;
    END IF;
    INSERT INTO Lines VALUES (V1, NAME);
    TRIG_INSERT_Lines(V1);
END;

-- ------------------------------------------------------
-- Procedure TO INSERT ON Stations
-- ------------------------------------------------------
CREATE OR REPLACE PROCEDURE INSERT_STATIONS (NAME
Stations.Station_Name%TYPE, Line Lines.Line_Name%TYPE, AtEnd NUMBER)
AS
    STATION_NOT_UNIQUE EXCEPTION;
    STATID Stations.Station_ID%TYPE;
    VAL NUMBER;
    LINEID Lines.Line_ID%TYPE;
BEGIN
    SELECT Line_ID INTO LINEID FROM Lines WHERE LOWER(Line_Name) =
LOWER(Line);
    SELECT COUNT(*) INTO VAL FROM Stations WHERE LOWER(Station_Name) =
LOWER(NAME);
    IF VAL > 0 THEN
        SELECT Line_ID INTO VAL FROM Stations WHERE LOWER(Station_Name) =
LOWER(NAME);
        IF VAL <> LINEID THEN
            INSERT_LINEXY(NAME, Line, AtEnd);
        ELSE
            RAISE STATION_NOT_UNIQUE;
        END IF;
    END IF;
    -- GET VALUE OF Station_ID
    SELECT COUNT(*) INTO VAL FROM Stations;
    IF VAL > 0 THEN
        SELECT MAX(Station_ID)+1 INTO STATID FROM Stations;
```

```
    ELSE
        STATID := 1;
    END IF;
    INSERT INTO Stations VALUES (STATID, NAME, LINEID);
    -- INSERT INTO LINE_LINE_ID
    INSERT_LINEXY(NAME, Line, AtEnd);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-120048, 'INVALID LINE NAME');
    WHEN STATION_NOT_UNIQUE THEN
        RAISE_APPLICATION_ERROR(-120049, 'STATION NAME NOT UNIQUE');
END;

-- ------------------------------------------------------
-- PROCEDURE TO INSERT ON Routes
-- ------------------------------------------------------
CREATE OR REPLACE PROCEDURE INSERT_ROUTE (SOURCE
Stations.Station_Name%TYPE, DEST Stations.Station_Name%TYPE)
AS
    SOURCEID Stations.STATION_ID%TYPE;
    DESTID Stations.STATION_ID%TYPE;
    SAME_STATION EXCEPTION;
    VAL NUMBER;
BEGIN
    SELECT Station_ID INTO SOURCEID FROM Stations WHERE LOWER(Station_Name)
= LOWER(SOURCE);
    SELECT Station_ID INTO DESTID FROM Stations WHERE LOWER(Station_Name) =
LOWER(DEST);
    IF SOURCEID <> DESTID THEN
        SELECT COUNT(*) INTO VAL FROM Routes;
        IF VAL > 0 THEN
            SELECT MAX(Route_ID) INTO VAL FROM Routes;
        ELSE
            VAL := 1;
        END IF;
        INSERT INTO Routes VALUES(VAL, SOURCEID, DESTID);
    ELSE
        RAISE SAME_STATION;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-120031,'INVALID STATION NAME');
    WHEN SAME_STATION THEN
        RAISE_APPLICATION_ERROR(-120032,'SOURCE AND DESTINATION STATIONS
CANNOT BE THE SAME');
END;
```